

Cours d'Informatique

1ère année SMP/SMC

2007/2008, Semestre 2

Mouad BEN MAMOUN

Moulay Driss RAHMANI

Département de Mathématiques et d'Informatique,
Université Mohammed V

ben_mamoun@fsr.ac.ma

mrahmani@fsr.ac.ma

Slide 1

I1 INDIANS; 24/03/2008

I2 INDIANS; 24/03/2008

Objectif et plan du cours

- **Objectif:**

- Apprendre les concepts de base de l'algorithmique et de la programmation
- Etre capable de mettre en oeuvre ces concepts pour analyser des problèmes simples et écrire les programmes correspondants

- **Plan:**

- Généralités (matériel d'un ordinateur, systèmes d'exploitation, langages de programmation, ...)
- Algorithmique (affectation, instructions conditionnelles, instructions itératives, fonctions, procédures, ...)
- MAPLE (un outil de programmation)

Informatique?

- Techniques du traitement **automatique** de l'**information** au moyen des ordinateurs
- Éléments d'un système informatique



Matériel: Principaux éléments d'un PC

- Unité centrale (le boîtier)
 - Processeur ou CPU (*Central Processing Unit*)
 - Mémoire centrale
 - Disque dur, lecteur disquettes, lecteur CD-ROM
 - Cartes spécialisées (cartes vidéo, réseau, ...)
 - Interfaces d'entrée-sortie (Ports série/parallèle, ...)
- Périphériques
 - Moniteur (l'écran), clavier, souris
 - Modem, imprimante, scanner, ...

Qu'est ce qu'un système d'exploitation?

- Ensemble de programmes qui gèrent le matériel et contrôlent les applications
 - Gestion des périphériques (affichage à l'écran, lecture du clavier, pilotage d'une imprimante, ...)
 - Gestion des utilisateurs et de leurs données (comptes, partage des ressources, gestion des fichiers et répertoires, ...)
 - Interface avec l'utilisateur (textuelle ou graphique):
Interprétation des commandes
 - Contrôle des programmes (découpage en tâches, partage du temps processeur, ...)

Langages informatiques

- Un langage informatique est un outil permettant de donner des ordres (**instructions**) à la machine
 - A chaque instruction correspond une action du processeur
- Intérêt : écrire des **programmes** (suite consécutive d'instructions) destinés à effectuer une tâche donnée
 - Exemple: un programme de gestion de comptes bancaires
- Contrainte: être compréhensible par la machine

Langage machine

- Langage **binaire**: l'information est exprimée et manipulée sous forme d'une suite de bits
- Un **bit** (*binary digit*) = 0 ou 1 (2 états électriques)
- Une combinaison de 8 bits = 1 **Octet** → $2^8 = 256$ possibilités qui permettent de coder tous les caractères alphabétiques, numériques, et symboles tels que ?, *, &, ...
 - Le code **ASCII** (*American Standard Code for Information Interchange*) donne les correspondances entre les caractères alphanumériques et leurs représentation binaire, Ex. A = 01000001, ? = 00111111
- Les opérations logiques et arithmétiques de base (addition, multiplication, ...) sont effectuées en binaire

L'assembleur

- Problème: le langage machine est difficile à comprendre par l'humain
- Idée: trouver un langage compréhensible par l'homme qui sera ensuite converti en langage machine
 - **Assembleur** (1er langage): exprimer les instructions élémentaires de façon symbolique

ADD A, 4
LOAD B
MOV A, OUT

traducteur → langage machine

...

- +: déjà plus accessible que le langage machine
- -: dépend du type de la machine (n'est pas **portable**)
- -: pas assez efficace pour développer des applications complexes

⇒ **Apparition des langages évolués**

Langages haut niveau

- Intérêts multiples pour le haut niveau:
 - proche du langage humain «anglais» (compréhensible)
 - permet une plus grande portabilité (indépendant du matériel)
 - Manipulation de données et d'expressions complexes (réels, objets, $a*b/c$, ...)
- Nécessité d'un traducteur (compilateur/interpréteur),
exécution plus ou moins lente selon le traducteur



Compilateur/interpréteur

- Compilateur: traduire le programme entier une fois pour toutes



- + plus rapide à l'exécution
 - + sécurité du code source
 - - il faut recompiler à chaque modification
- Interpréteur: traduire au fur et à mesure les instructions du programme à chaque exécution

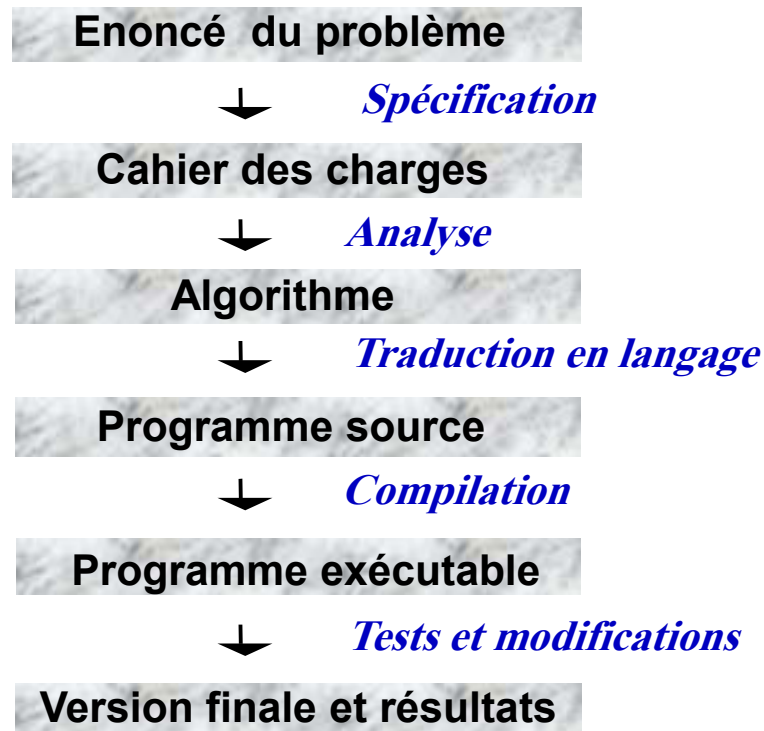


- + exécution instantanée appréciable pour les débutants
- - exécution lente par rapport à la compilation

Langages de programmation:

- Deux types de langages:
 - Langages procéduraux
 - Langages orientés objets
- Exemples de langages:
 - **Fortran, Cobol, Pascal, C, ...**
 - **C++, Java, ...**
- Choix d'un langage?

Etapes de réalisation d'un programme



La réalisation de programmes passe par l'écriture d'algorithmes
⇒ D'où l'intérêt de l'**Algorithmique**

Algorithmique

- Le terme **algorithme** vient du nom du mathématicien arabe **Al-Khawarizmi** (820 après J.C.)
- Un algorithme est une description complète et détaillée des actions à effectuer et de leur séquençement pour arriver à un résultat donné
 - Intérêt: séparation analyse/codage (pas de préoccupation de syntaxe)
 - Qualités: **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), ...
- **L'algorithmique** désigne aussi la discipline qui étudie les algorithmes et leurs applications en Informatique
- Une bonne connaissance de l'algorithmique permet d'écrire des algorithmes exacts et efficaces

Représentation d'un algorithme

Historiquement, deux façons pour représenter un algorithme:

- **L'Organigramme:** représentation graphique avec des symboles (carrés, losanges, etc.)
 - offre une vue d'ensemble de l'algorithme
 - représentation quasiment abandonnée aujourd'hui
- **Le pseudo-code:** représentation textuelle avec une série de conventions ressemblant à un langage de programmation (sans les problèmes de syntaxe)
 - plus pratique pour écrire un algorithme
 - représentation largement utilisée

Algorithmique

Notions et instructions de base

Notion de variable

- Dans les langages de programmation une **variable** sert à stocker la valeur d'une donnée
- Une variable désigne en fait un emplacement mémoire dont le contenu peut changer au cours d'un programme (d'où le nom variable)
- Règle : Les variables doivent être **déclarées** avant d'être utilisées, elle doivent être caractérisées par :
 - un nom (**Identificateur**)
 - un **type** (entier, réel, caractère, chaîne de caractères, ...)

Choix des identificateurs (1)

Le choix des noms de variables est soumis à quelques règles qui varient selon le langage, mais en général:

- Un nom doit commencer par une lettre alphabétique
exemple valide: A1 **exemple invalide: 1A**
- doit être constitué uniquement de lettres, de chiffres et du soulignement _ (Eviter les caractères de ponctuation et les espaces)
valides: SMIP2007, SMP_2007 **invalides: SMP 2005, SMI-2007, SMP;2007**
- doit être différent des mots réservés du langage (par exemple en Java: **int, float, else, switch, case, default, for, main, return, ...**)
- La longueur du nom doit être inférieure à la taille maximale spécifiée par le langage utilisé

Choix des identificateurs (2)

Conseil: pour la lisibilité du code choisir des noms significatifs qui décrivent les données manipulées

exemples: TotalVentes2004, Prix_TTC, Prix_HT

Remarque: en pseudo-code algorithmique, on va respecter les règles citées, même si on est libre dans la syntaxe

Types des variables

Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre, les types offerts par la plus part des langages sont:

- Type numérique (entier ou réel)
 - **Byte** (codé sur 1 octet): de 0 à 255
 - **Entier court** (codé sur 2 octets) : -32 768 à 32 767
 - **Entier long** (codé sur 4 ou 8 octets)
 - **Réel simple précision** (codé sur 4 octets)
 - **Réel double précision** (codé sur 8 octets)
- Type logique ou booléen: deux valeurs VRAI ou FAUX
- Type caractère: lettres majuscules, minuscules, chiffres, symboles, ...
exemples: 'A', 'a', '1', '?', ...
- Type chaîne de caractère: toute suite de caractères,
exemples: " Nom, Prénom", "code postale: 1000", ...

Déclaration des variables

- Rappel: toute variable utilisée dans un programme doit avoir fait l'objet d'une déclaration préalable
- En pseudo-code, on va adopter la forme suivante pour la déclaration de variables

Variables liste d'identificateurs : type

- Exemple:

Variables i, j, k : entier

x, y : réel

OK: booléen

ch1, ch2 : chaîne de caractères

- Remarque: pour le type numérique on va se limiter aux entiers et réels sans considérer les sous types

L'instruction d'affectation

- **l'affectation** consiste à attribuer une valeur à une variable (ça consiste en fait à remplir ou à modifier le contenu d'une zone mémoire)
- En pseudo-code, l'affectation se note avec le signe \leftarrow
 $\text{Var} \leftarrow e$: attribue la valeur de e à la variable Var
 - e peut être une valeur, une autre variable ou une expression
 - Var et e doivent être de même type ou de types compatibles
 - l'affectation ne modifie que ce qui est à gauche de la flèche
- **Ex valides:**

$i \leftarrow 1$	$j \leftarrow i$	$k \leftarrow i+j$
$x \leftarrow 10.3$	$\text{OK} \leftarrow \text{FAUX}$	$\text{ch1} \leftarrow \text{"SMI"}$
$\text{ch2} \leftarrow \text{ch1}$	$x \leftarrow 4$	$x \leftarrow j$

(voir la déclaration des variables dans le transparent précédent)
- **non valides:** $i \leftarrow 10.3$ $\text{OK} \leftarrow \text{"SMI"}$ $j \leftarrow x$

Quelques remarques

- Beaucoup de langages de programmation (C/C++, Java, ...) utilisent le signe égal = pour l'affectation \leftarrow . Attention aux confusions:
 - l'affectation n'est pas commutative : $A=B$ est différente de $B=A$
 - l'affectation est différente d'une équation mathématique :
 - $A=A+1$ a un sens en langages de programmation
 - $A+1=2$ n'est pas possible en langages de programmation et n'est pas équivalente à $A=1$
- Certains langages donnent des valeurs par défaut aux variables déclarées. Pour éviter tout problème il est préférable **d'initialiser les variables** déclarées

Exercices simples sur l'affectation (1)

Donnez les valeurs des variables A, B et C après exécution des instructions suivantes ?

Variables A, B, C: Entier

Début

$A \leftarrow 3$

$B \leftarrow 7$

$A \leftarrow B$

$B \leftarrow A+5$

$C \leftarrow A + B$

$C \leftarrow B - A$

Fin

Exercices simples sur l'affectation (2)

Donnez les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B : Entier

Début

$A \leftarrow 1$

$B \leftarrow 2$

$A \leftarrow B$

$B \leftarrow A$

Fin

Les deux dernières instructions permettent-elles d'échanger les valeurs de A et B ?

Exercices simples sur l'affectation (3)

Ecrire un algorithme permettant d'échanger les valeurs de deux variables A et B

Expressions et opérateurs

- Une **expression** peut être une valeur, une variable ou une opération constituée de variables reliées par des **opérateurs**
exemples: 1, b, a*2, a+ 3*b-c, ...
- L'évaluation de l'expression fournit une valeur unique qui est le résultat de l'opération
- Les **opérateurs** dépendent du type de l'opération, ils peuvent être :
 - des opérateurs arithmétiques: +, -, *, /, % (modulo), ^ (puissance)
 - des opérateurs logiques: NON, OU, ET
 - des opérateurs relationnels: =, ≠, <, >, <=, >=
 - des opérateurs sur les chaînes: & (concaténation)
- Une expression est évaluée de gauche à droite mais en tenant compte de **priorités**

Priorité des opérateurs

- Pour les opérateurs arithmétiques donnés ci-dessus, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

- $^$: (élévation à la puissance)
- $*$, $/$ (multiplication, division)
- $\%$ (modulo)
- $+$, $-$ (addition, soustraction)

exemple: $2 + 3 * 7$ vaut 23

- En cas de besoin (ou de doute), on utilise les parenthèses pour indiquer les opérations à effectuer en priorité

exemple: $(2 + 3) * 7$ vaut 35

Les instructions d'entrées-sorties: lecture et écriture (1)

- Les instructions de lecture et d'écriture permettent à la machine de communiquer avec l'utilisateur
- La **lecture** permet d'entrer des données à partir du clavier
 - En pseudo-code, on note: **lire (var)**
la machine met la valeur entrée au clavier
dans la zone mémoire nommée var
 - Remarque: Le programme s'arrête lorsqu'il rencontre une instruction Lire et ne se poursuit qu'après la frappe d'une valeur au clavier et de la touche Entrée

Les instructions d'entrées-sorties: lecture et écriture (2)

- **L'écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
 - En pseudo-code, on note: **écrire (var)**
la machine affiche le contenu de la zone mémoire var
 - Conseil: Avant de lire une variable, il est fortement conseillé d'écrire des messages à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper

Exemple (lecture et écriture)

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui calcule et affiche le double de ce nombre

Algorithme Calcul_double

variables A, B : **entier**

Début

écrire("entrer le nombre ")

lire(A)

$B \leftarrow 2 * A$

écrire("le double de ", A, "est :", B)

Fin

Exercice (lecture et écriture)

Ecrire un algorithme qui vous demande de saisir votre nom puis votre prénom et qui affiche ensuite votre nom complet

Algorithme AffichageNomComplet

variables Nom, Prenom, Nom_Complet : **chaîne de caractères**

Début

écrire("entrez votre nom")

lire(Nom)

écrire("entrez votre prénom")

lire(Prenom)

 Nom_Complet \leftarrow Nom & Prenom

écrire("Votre nom complet est : ", Nom_Complet)

Fin

Tests: instructions conditionnelles (1)

- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée
- On utilisera la forme suivante: **Si condition alors**
instruction ou suite d'instructions1
Sinon
instruction ou suite d'instructions2
Finsi
 - la condition ne peut être que vraie ou fausse
 - si la condition est vraie, se sont les instructions1 qui seront exécutées
 - si la condition est fausse, se sont les instructions2 qui seront exécutées
 - la condition peut être une condition simple ou une condition composée de plusieurs conditions

Tests: instructions conditionnelles (2)

- La partie Sinon n'est pas obligatoire, quand elle n'existe pas et que la condition est fausse, aucun traitement n'est réalisé
 - On utilisera dans ce cas la forme simplifiée suivante:

Si condition alors

instruction ou suite d'instructions1

Finsi

Exemple (Si...Alors...Sinon)

Algorithme AffichageValeurAbsolue (version1)

Variable x : réel

Début

Ecrire (" Entrez un réel : “)

Lire (x)

Si ($x < 0$) **alors**

Ecrire ("la valeur absolue de ", x , "est:", $-x$)

Sinon

Ecrire ("la valeur absolue de ", x , "est:", x)

Finsi

Fin

Exemple (Si...Alors)

Algorithme AffichageValeurAbsolue (version2)

Variable x,y : réel

Début

Ecrire (" Entrez un réel : “)

Lire (x)

$y \leftarrow x$

Si ($x < 0$) **alors**

$y \leftarrow -x$

Finsi

Ecrire ("la valeur absolue de ", x, "est:",y)

Fin

Exercice (tests)

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui teste et affiche s'il est divisible par 3

Algorithme Divisible_par3

Variable n : entier

Début

Ecrire " Entrez un entier : "

Lire (n)

Si ($n \% 3 = 0$) **alors**

Ecrire (n, " est divisible par 3")

Sinon

Ecrire (n, " n'est pas divisible par 3")

Finsi

Fin

Conditions composées

- Une condition composée est une condition formée de plusieurs conditions simples reliées par des opérateurs logiques:
ET, OU, OU exclusif (XOR) et NON
- Exemples :
 - x compris entre 2 et 6 : $(x > 2) \text{ ET } (x < 6)$
 - n divisible par 3 ou par 2 : $(n \% 3 = 0) \text{ OU } (n \% 2 = 0)$
 - deux valeurs et deux seulement sont identiques parmi a, b et c :
 $(a=b) \text{ XOR } (a=c) \text{ XOR } (b=c)$
- L'évaluation d'une condition composée se fait selon des règles présentées généralement dans ce qu'on appelle tables de vérité

Tables de vérité

C1	C2	C1 ET C2
VRAI	VRAI	VRAI
VRAI	FAUX	FAUX
FAUX	VRAI	FAUX
FAUX	FAUX	FAUX

C1	C2	C1 OU C2
VRAI	VRAI	VRAI
VRAI	FAUX	VRAI
FAUX	VRAI	VRAI
FAUX	FAUX	FAUX

C1	C2	C1 XOR C2
VRAI	VRAI	FAUX
VRAI	FAUX	VRAI
FAUX	VRAI	VRAI
FAUX	FAUX	FAUX

C1	NON C1
VRAI	FAUX
FAUX	VRAI

Tests imbriqués

- Les tests peuvent avoir un degré quelconque d'imbrications

Si condition1 alors

Si condition2 alors

instructionsA

Sinon

instructionsB

Finsi

Sinon

Si condition3 alors

instructionsC

Finsi

Finsi

Tests imbriqués: exemple (version 1)

Variable n : entier

Début

Ecrire ("entrez un nombre : ")

Lire (n)

Si ($n < 0$) **alors**

Ecrire ("Ce nombre est négatif")

Sinon

Si ($n = 0$) **alors**

Ecrire ("Ce nombre est nul")

Sinon

Ecrire ("Ce nombre est positif")

Finsi

Finsi

Fin

Tests imbriqués: exemple (version 2)

Variable n : entier

Début

Ecrire ("entrez un nombre : ")

Lire (n)

Si ($n < 0$) **alors** Ecrire ("Ce nombre est négatif")

Finsi

Si ($n = 0$) **alors** Ecrire ("Ce nombre est nul")

Finsi

Si ($n > 0$) **alors** Ecrire ("Ce nombre est positif")

Finsi

Fin

Remarque : dans la version 2 on fait trois tests systématiquement alors que dans la version 1, si le nombre est négatif on ne fait qu'un seul test

Conseil : utiliser les tests imbriqués pour limiter le nombre de tests et placer d'abord les conditions les plus probables

Tests imbriqués: exercice

Le prix de photocopies dans une reprographie varie selon le nombre demandé: 0,5 DH la copie pour un nombre de copies inférieur à 10, 0,4DH pour un nombre compris entre 10 et 20 et 0,3DH au-delà.

Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées, qui calcule et affiche le prix à payer

Tests imbriqués: corrigé de l'exercice

Variables copies : entier
prix : réel

Début

Ecrire ("Nombre de photocopies : ")

Lire (copies)

Si (copies < 10) **Alors**

 prix ← copies*0.5

Sinon Si (copies) < 20

 prix ← copies*0.4

Sinon

 prix ← copies*0.3

Finsi

Finsi

Ecrire ("Le prix à payer est : ", prix)

Fin

Instructions itératives: les boucles

- Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois
- On distingue trois sortes de boucles en langages de programmation :
 - Les **boucles tant que** : on y répète des instructions tant qu'une certaine condition est réalisée
 - Les **boucles jusqu'à** : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
 - Les **boucles pour** ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

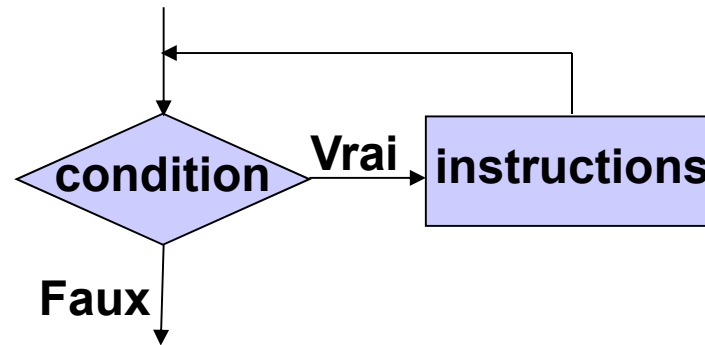
(Dans ce cours, on va s'intéresser essentiellement aux boucles *Tant que* et boucles *Pour* qui sont plus utilisées et qui sont définies en Maple)

Les boucles Tant que

TantQue (condition)

instructions

FinTantQue



- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, ...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après FinTantQue

Les boucles Tant que : remarques

- Le nombre d'itérations dans une boucle TantQue n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment

⇒ **Attention aux boucles infinies**

- Exemple de boucle infinie :

$i \leftarrow 2$

TantQue ($i > 0$)

$i \leftarrow i+1$ (attention aux erreurs de frappe : + au lieu de -)

FinTantQue

Boucle Tant que : exemple1

Contrôle de saisie d'une lettre majuscule jusqu'à ce que le caractère entré soit valable

Variable C : caractère

Debut

Ecrire (" Entrez une lettre majuscule ")

Lire (C)

TantQue (C < 'A' ou C > 'Z')

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

FinTantQue

Ecrire ("Saisie valable")

Fin

Boucle Tant que : exemple2

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

version 1

Variables som, i : entier

Debut

$i \leftarrow 0$

$som \leftarrow 0$

TantQue (som <=100)

$i \leftarrow i+1$

$som \leftarrow som+i$

FinTantQue

 Ecrire (" La valeur cherchée est N= ", i)

Fin

Boucle Tant que : exemple2 (version2)

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

version 2: attention à l'ordre des instructions et aux valeurs initiales

Variables som, i : entier

Debut

som \leftarrow 0

i \leftarrow 1

TantQue (som \leq 100)

som \leftarrow som + i

i \leftarrow i+1

FinTantQue

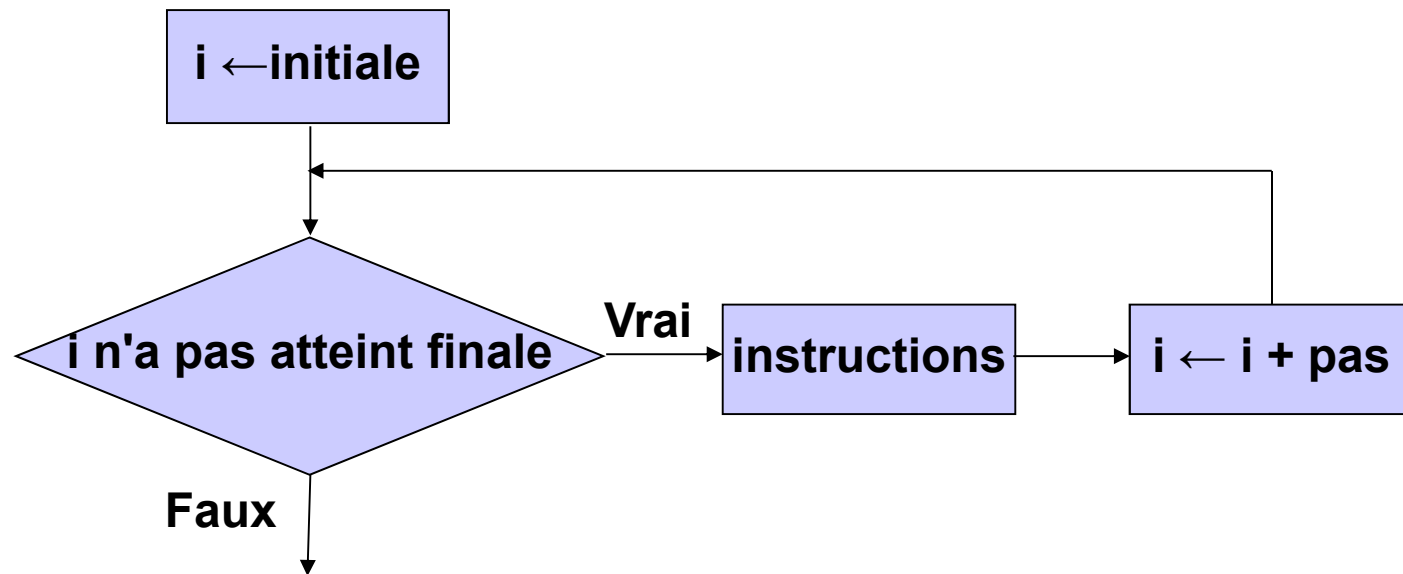
Ecrire (" La valeur cherchée est N= ", i-1)

Fin

Les boucles Pour

Pour compteur **allant de** initiale à finale par **pas** valeur du pas
instructions

FinPour



Les boucles Pour

- Remarque : le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle
- **Compteur** est une variable de type entier (ou caractère). Elle doit être déclarée
- **Pas** est un entier qui peut être positif ou négatif. **Pas** peut ne pas être mentionné, car par défaut sa valeur est égal à 1. Dans ce cas, le nombre d'itérations est égal à finale - initiale + 1
- **Initiale et finale** peuvent être des valeurs, des variables définies avant le début de la boucle ou des expressions de même type que compteur

Déroulement des boucles Pour

- 1) La valeur initiale est affectée à la variable compteur
- 2) On compare la valeur du compteur et la valeur de finale :
 - a) Si la valeur du compteur est $>$ à la valeur finale dans le cas d'un pas positif (ou si compteur est $<$ à finale pour un pas négatif), on sort de la boucle et on continue avec l'instruction qui suit FinPour
 - b) Si compteur est \leq à finale dans le cas d'un pas positif (ou si compteur est \geq à finale pour un pas négatif), instructions seront exécutées
 - i. Ensuite, la valeur de compteur est incrémentée de la valeur du pas si pas est positif (ou décrémente si pas est négatif)
 - ii. On recommence l'étape 2 : La comparaison entre compteur et finale est de nouveau effectuée, et ainsi de suite ...

Boucle Pour : exemple1

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul

Variables x , $puiss$: réel
 n , i : entier

Debut

Ecrire (" Entrez la valeur de x ")

Lire (x)

Ecrire (" Entrez la valeur de n ")

Lire (n)

$puiss \leftarrow 1$

Pour i allant de 1 à n

$puiss \leftarrow puiss * x$

FinPour

Ecrire (x , " à la puissance ", n , " est égal à ", $puiss$)

Fin

Boucle Pour : exemple1 (version 2)

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul (**version 2 avec un pas négatif**)

Variables x , $puiss$: réel
 n , i : entier

Debut

Ecrire (" Entrez respectivement les valeurs de x et n ")

Lire (x , n)

$puiss \leftarrow 1$

Pour i allant de n à 1 par pas -1

$puiss \leftarrow puiss * x$

FinPour

Ecrire (x , " à la puissance ", n , " est égal à ", $puiss$)

Fin

Boucle Pour : remarque

- Il faut éviter de modifier la valeur du compteur (et de finale) à l'intérieur de la boucle. En effet, une telle action :
 - perturbe le nombre d'itérations prévu par la boucle Pour
 - rend difficile la lecture de l'algorithme
 - présente le risque d'aboutir à une boucle infinie

Exemple : **Pour** i allant de 1 à 5

$i \leftarrow i - 1$

écrire(" i = ", i)

Finpour

Lien entre Pour et TantQue

La boucle Pour est un cas particulier de Tant Que (cas où le nombre d'itérations est connu et fixé) . Tout ce qu'on peut écrire avec Pour peut être remplacé avec TantQue (la réciproque est fausse)

Pour compteur **allant de** initiale à finale par **pas** valeur du pas
instructions

FinPour

peut être remplacé par :
(cas d'un pas positif)

compteur \leftarrow initiale
TantQue compteur \leq finale
instructions
compteur \leftarrow compteur+pas
FinTantQue

Lien entre Pour et TantQue: exemple

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul (**version avec TantQue**)

Variables x , puiss : réel
 n , i : entier

Debut

Ecrire (" Entrez la valeur de x ")
Lire (x)
Ecrire (" Entrez la valeur de n ")
Lire (n)

$\text{puiss} \leftarrow 1$

$i \leftarrow 1$

TantQue ($i \leq n$)

$\text{puiss} \leftarrow \text{puiss} * x$
 $i \leftarrow i + 1$

FinTantQue

Ecrire (x , " à la puissance ", n , " est égal à ", puiss)

Fin

Boucles imbriquées

- Les instructions d'une boucle peuvent être des instructions itératives. Dans ce cas, on aboutit à des **boucles imbriquées**

- **Exemple:**

```
Pour i allant de 1 à 5
    Pour j allant de 1 à i
        écrire("O")
    FinPour
    écrire("X")
FinPour
```

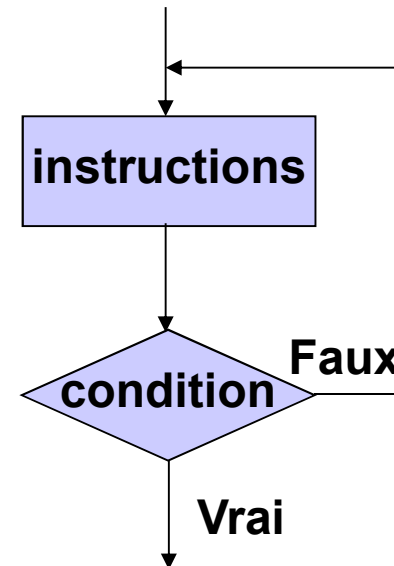
Exécution

Les boucles Répéter ... jusqu'à ...

Répéter

instructions

Jusqu'à condition



- Condition est évaluée après chaque itération
- les instructions entre *Répéter* et *jusqu'à* sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que condition soit vraie (tant qu'elle est fausse)

Boucle Répéter jusqu'à : exemple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (**version avec répéter jusqu'à**)

Variables som, i : entier

Debut

som \leftarrow 0

i \leftarrow 0

Répéter

i \leftarrow i+1

som \leftarrow som+i

Jusqu'à (som > 100)

Ecrire (" La valeur cherchée est N= ", i)

Fin

Choix d'un type de boucle

- Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser *la boucle Pour*
- S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des *boucles TantQue* ou *répéter jusqu'à*
- Pour le choix entre *TantQue* et *jusqu'à* :
 - Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera *TantQue*
 - Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera *répéter jusqu'à*

MAPLE

Présentation générale et syntaxe des instructions de base

Maple

- Maple est un logiciel de calcul formel et numérique
 - **Calcul formel** : calcul sur des expressions littérales sans évaluation numérique (Maple peut calculer des dérivées, des intégrales, des développements limités, ...)

$$> \text{int}(1-x+x^3,x); \quad \longrightarrow \quad x - \frac{x^2}{2} + \frac{x^4}{4}$$

$$> \text{taylor}(\sin(x),x=0,6); \quad \longrightarrow \quad x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^6)$$

- **Calcul numérique** : calcul sur des valeurs (avec une grande précision)

$$> 30!; \quad \longrightarrow \quad 265252859812191058636308480000000$$

$$> \text{evalf}(\text{sqrt}(2),50); \quad \longrightarrow \quad 1.414213562373095048801688$$

Maple : les packages

- Maple dispose d'un certain nombre de **packages** (bibliothèques). Chacun de ces packages est spécialisé dans un traitement particulier. Comme exemples de ces packages, on a :
 - **linalg** : pour l'algèbre linéaire
 - **plots** : pour le tracé des courbes
 - **geometry** : pour la géométrie
 - **student** : ce package est conçu pour assister l'enseignement des mathématiques de base (intéressant pour les étudiants)
- Pour utiliser certaines commandes et fonctions de Maple, il faut d'abord charger le package qui les contient avec la commande **with** :
 - **with** (NomLibrairie) : charge le package NomLibrairie
 - **with** (NomLib, NomCmd) : charge la commande NomCmd du package NomLib

Maple : Généralités

- Chaque instruction Maple doit se terminer par ; ou :
 - Si l'instruction se termine par ; Maple l'exécute et affiche le résultat
 - Si l'instruction se termine par : Maple l'exécute sans afficher le résultat
- Pour introduire un texte en tant que **commentaire**, il suffit de précéder la ligne par # (le texte est alors ignoré par Maple)
- Il est aussi possible d'écrire des **commentaires** en cliquant sur l'icône **T** de la barre d'outils et sur l'icône **[>** pour revenir en mode normal
- Maple fait la distinction entre les lettres majuscules et minuscules (SMI, Smi, sml et smi sont différents pour Maple)

Maple : nom et type des variables

- Le **nom d'une variable** peut être une combinaison de lettres et de chiffres, mais qui commence par une lettre, qui ne contient pas d'espaces et qui est différente des mots réservés (commandes Maple)
- Le **type d'une variable** est attribué automatiquement par Maple selon le contexte (exemple : si A prend la valeur 2, A sera de type **integer**, si A prend la valeur \sqrt{A} sera de type **float**)
- Les principaux types définis en Maple sont : **integer** (entier), **float** (réel), **fraction** (rationnel), **complex** (complexe), **string** (chaîne de caractères), **boolean** (booléen), **array** (tableau), **matrix** (matrice)
- Maple offre quelques commandes relatifs aux types :
ex : **whattype**(var) donne le type de la variable var

Maple : l'affectation

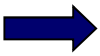
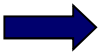
- Le symbole d'**affectation** ← se note en Maple avec **:=**

exemple : `i:= 1; j:= i+1;`

- Attention :** en Maple **a=b** n'est pas une instruction d'affectation, mais une expression de type logique (**boolean**) qui est vrai si les deux valeurs a et b sont égales et fausse sinon

- Maple n'évalue l'expression logique **a=b** que si on le demande explicitement. Pour cela, on utilisera la commande **evalb**

exemple : `a:= 1; b:= 2;`

<code>> a=b;</code>		<code>1=2</code>
<code>> evalb (a=b);</code>		<code>false</code>

Maple : instructions d'entrées-sorties

- **print(var)** permet d'afficher la valeur de la variable var (c'est l'équivalent de **écrire** en pseudo code). Si var n'a pas de valeur, Maple affiche le nom de la variable
- **print(`chaine`)** permet d'afficher la chaîne de caractères qui est entre ``
> a:=1: b:=2: print(`a vaut`,a, `et b vaut`,b);
➡ a vaut ,1 et b vaut, 2
- **readstat** permet de saisir des données à partir du clavier (c'est l'équivalent de **lire** en pseudo code)
 - **Syntaxe:** var:=**readstat**(`texte`) ➡ Maple affiche le texte entre `` et attend qu'on entre une valeur au clavier **qui doit être suivie de ; ou :**
> n:=readstat(`entrez la valeur de n : `);
- Remarque : il existe d'autres commandes pour les entrées-sorties en Maple

Maple : syntaxe des tests

Écriture en pseudo code

Si condition **alors**
 instructions
Finsi

Si condition **alors**
 instructions 1
Sinon
 instructions2
Finsi

Traduction en Maple

if condition **then**
 instructions
fi;

if condition **then**
 instructions1
else
 instructions2
fi;

Maple : syntaxe des boucles

Écriture en pseudo code

TantQue condition
 instructions
FinTantQue

Traduction en Maple

while condition **do**
 instructions
od;

Pour i allant de $v1$ à $v2$ par pas p
 instructions
FinPour

for i from $v1$ to $v2$ by p do
 instructions
od;

ALGORITHMIQUE

Fonctions et procédures

Fonctions et procédures

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :
 - permettent de "**factoriser**" les **programmes**, càd de mettre en commun les parties qui se répètent
 - permettent une **structuration** et une **meilleure lisibilité** des programmes
 - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
 - ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes

Fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat à partir des valeurs des paramètres**
- Une fonction s'écrit en dehors du programme principal sous la forme :

Fonction nom_fonction (paramètres et leurs types) : type_fonction

Instructions constituant le corps de la fonction

retourne ...

FinFonction

- Pour le choix d'un nom de fonction il faut respecter les mêmes règles que celles pour les noms de variables
- type_fonction est le type du résultat retourné
- L'instruction **retourne** sert à retourner la valeur du résultat

Fonctions : exemples

- La fonction SommeCarre suivante calcule la somme des carrées de deux réels x et y :

Fonction SommeCarre (x : réel, y: réel) : réel

variable z : réel

$z \leftarrow x^2 + y^2$

retourne (z)

FinFonction

- La fonction Pair suivante détermine si un nombre est pair :

Fonction Pair (n : entier) : booléen

retourne (n%2=0)

FinFonction

Utilisation des fonctions

- L'utilisation d'une fonction se fera par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, devra être affecté ou être utilisé dans une expression, une écriture, ...
- **Exepmle : Algorithme exepmleAppelFonction**
variables z : réel, b : booléen
Début
 b ← Pair(3)
 z ← 5*SommeCarre(7,2)+1
 écrire("SommeCarre(3,5)= ", SommeCarre(3,5))
Fin
- Lors de l'appel Pair(3) le **paramètre formel** n est remplacé par le **paramètre effectif** 3

Procédures

- Dans certains cas, on peut avoir besoin de répéter une tâche dans plusieurs endroits du programme, mais que dans cette tâche on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois
- Dans ces cas on ne peut pas utiliser une fonction, on utilise une **procédure**
- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**
- Une procédure s'écrit en dehors du programme principal sous la forme :

Procédure nom_procédure (paramètres et leurs types)

Instructions constituant le corps de la procédure

FinProcédure

- Remarque : une procédure peut ne pas avoir de paramètres

Appel d'une procédure

- L'appel d'une procédure, se fait dans le programme principale ou dans une autre procédure par une instruction indiquant le nom de la procédure :

Procédure exemple_proc (...)

...

FinProcédure

Algorithme exepmleAppelProcédure

Début

 exemple_proc (...)

...

Fin

- Remarque : contrairement à l'appel d'une fonction, on ne peut pas affecter la procédure appelée ou l'utiliser dans une expression. L'appel d'une procédure est une instruction autonome

Paramètres d'une procédure

- Les paramètres servent à échanger des données entre le programme principale (ou la procédure appelante) et la procédure appelée
- Les paramètres placés dans la déclaration d'une procédure sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)
- Les paramètres placés dans l'appel d'une procédure sont appelés **paramètres effectifs**. ils contiennent les valeurs pour effectuer le traitement
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre

Transmission des paramètres

Il existe deux modes de transmission de paramètres dans les langages de programmation :

- **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification
- **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la procédure
 - **Remarque** : le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse
- En pseudo-code, on va préciser explicitement le mode de transmission dans la déclaration de la procédure

Transmission des paramètres : exemples

Procédure incrementer1 (**x : entier par valeur, y : entier par adresse**)

$x \leftarrow x+1$

$y \leftarrow y+1$

FinProcédure

Algorithme Test_incrementer1

variables n, m : entier

Début

$n \leftarrow 3$

$m \leftarrow 3$

incrementer1(n, m)

écrire (" n= ", n, " et m= ", m)

Fin

résultat :

Remarque : l'instruction $x \leftarrow x+1$ n'a pas de sens avec un passage par valeur

Transmission par valeur, par adresse : exemples

Procédure qui calcule la somme et le produit de deux entiers :

Procédure SommeProduit (x,y: entier **par valeur**, som, prod : entier **par adresse**)

 som \leftarrow x+y

 prod \leftarrow x*y

FinProcédure

Procédure qui échange le contenu de deux variables :

Procédure Echange (**x : réel par adresse**, **y : réel par adresse**)

variables z : réel

 z \leftarrow x

 x \leftarrow y

 y \leftarrow z

FinProcédure

Variables locales et globales (1)

- On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie")
- Une **variable locale** n'est connue qu'à l'intérieur du module ou elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution
- Une **variable globale** est connue par l'ensemble des modules et le programme principale. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme

Variables locales et globales (2)

- La manière de distinguer la déclaration des variables locales et globales diffère selon le langage
 - En général, les variables déclarées à l'intérieur d'une fonction ou procédure sont considérées comme variables locales
- En pseudo-code, on va adopter cette règle pour les variables locales et on déclarera les variables globales dans le programme principale
- **Conseil :** Il faut utiliser autant que possible des variables locales plutôt que des variables globales. Ceci permet d'économiser la mémoire et d'assurer l'indépendance de la procédure ou de la fonction

Fonctions et procédures en Maple (1)

- En Maple, il n'y a pas de distinction entre les notions de fonction et procédure. Les deux se déclarent de la même façon comme suit :

```
identificateur:= proc (paramètres)  
    local     $l_1, \dots, l_n$  ;  
    global  $g_1, \dots, g_k$  ;  
    instructions  
    résultat  
end;
```

- Identificateur est le nom de la fonction ou de la procédure
- En Maple, on précise explicitement si les variables sont locales ou globales par les mots clés **local** et **global**

Fonctions et procédures en Maple (2)

- Une variable globale est connue en dehors de la procédure où elle a été définie dans l'ensemble de la session de calcul
- Les paramètres, les variables locales et globales sont facultatifs, ils peuvent ne pas figurer dans la déclaration
- Une procédure Maple peut rendre un seul résultat (comme une fonction), plusieurs résultats ou aucun résultat
- Pour rendre plusieurs résultats, on peut utiliser une liste, un ensemble, un tableau (on verra ces structures la séance prochaine)
- Le résultat de la procédure est donné soit implicitement par la dernière instruction, soit par la commande RETURN
- RETURN (v_1, \dots, v_n) arrête le déroulement de la procédure et renvoie les valeurs de v_1, \dots, v_n sous forme d'une séquence

Procédures Maple : remarques

- Maple interdit la modification de la valeur d'un paramètre à l'intérieur d'une procédure (pas de transmission par adresse)
- Après **end;** Maple affiche le texte de la procédure. Dans le cas où **end** est suivi de : rien n'est affiché

```
> carre:=proc(x,y)
> x^2+y^2;
> end;          carre:=proc (x, y) x^2+y^2 end proc
```

- En Maple, une procédure peut être appelée sans être affectée. Elle peut aussi être affectée à une variable

```
> carre(1,2);          5
> a:=carre(3,3);       a := 18
```

Procédures Maple : exemples (1)

```
> exemple:=proc(a,b)
> local c,d,e;
> c:=a+b; d:=a-b; e:=a*b;
> RETURN(c,d,e);
> d:=c+e;
> end;
```

```
> exemple(4,7);      ➡      11, -3, 28
```

Remarque : l'exécution s'arrête après RETURN. L'instruction `d:=c+e` n'est pas exécutée, le résultat est donné sous forme d'une séquence

Procédures Maple : exemples (2)

Exemple : procédure qui calcule la somme des n premiers entiers

```
> somme:=proc()  
> local n,i,som;  
> som:=0;  
> n:=readstat(`entrez la valeur de n : `);  
> for i from 1 to n do  
> som:=som+i;  
> od;  
> print(`somme=`,som);  
> end;
```

```
> somme();
```

sur l'écran apparaît le message :

entrez la valeur de n :

si on entre 3, on obtient **somme=,6**

Récurtivité

- Un module (fonction ou procédure) peut s'appeler lui-même: on dit que c'est un module **récuratif**
- Tout module récuratif doit posséder un cas limite (cas trivial) qui arrête la récurativité
- Exemple : Calcul du factorielle

```
Fonction fact (n : entier ) : entier
    Si (n=0) alors
        retourne (1)
    Sinon
        retourne (n*fact(n-1))
    Finsi
FinFonction
```

Fonctions récursives : exercice

- Ecrivez une fonction récursive (puis itérative) qui calcule le terme n de la suite de Fibonacci définie par :
$$U(0)=U(1)=1$$
$$U(n)=U(n-1)+U(n-2)$$

```
Fonction Fib (n : entier ) : entier
    Variable res : entier
    Si (n=1 OU n=0) alors
        res ← 1
    Sinon
        res ← Fib(n-1)+Fib(n-2)
    Finsi
    retourne (res)
FinFonction
```

Fonctions récursives : exercice (suite)

- Une fonction itérative pour le calcul de la suite de Fibonacci :

Fonction Fib (n : entier) : entier

Variables i, AvantDernier, Dernier, Nouveau : entier

Si (n=1 OU n=0) **alors** **retourne** (1)

Finsi

AvantDernier ← 1, Dernier ← 1

Pour i allant de 2 à n

 Nouveau ← Dernier + AvantDernier

 AvantDernier ← Dernier

 Dernier ← Nouveau

FinPour

retourne (Nouveau)

FinFonction

Remarque: la solution récursive est plus facile à écrire

Procédures récursives : exemple

- Une procédure récursive qui permet d'afficher la valeur binaire d'un entier n

Procédure binaire (n : entier)

Si (n<>0) **alors**

 binaire (n/2)

 écrire (n mod 2)

Finsi

FinProcédure

ALGORITHMIQUE

Les tableaux

Exemple introductif

- Supposons qu'on veut conserver les notes d'une classe de 30 étudiants pour extraire quelques informations. Par exemple : calcul du nombre d'étudiants ayant une note supérieure à 10
- Le seul moyen dont nous disposons actuellement consiste à déclarer 30 variables, par exemple **N1**, ..., **N30**. Après 30 instructions lire, on doit écrire 30 instructions Si pour faire le calcul

nbre ← 0

Si (N1 >10) alors nbre ←nbre+1 FinSi

....

Si (N30>10) alors nbre ←nbre+1 FinSi

c'est lourd à écrire

- Heureusement, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans **une seule structure de donnée** appelée **tableau**

Tableaux

- Un **tableau** est un ensemble d'éléments de même type désignés par un identificateur unique
- Une variable entière nommée **indice** permet d'indiquer la position d'un élément donné au sein du tableau et de déterminer sa valeur
- La **déclaration** d'un tableau s'effectue en précisant le **type** de ses éléments et sa **dimension** (le nombre de ses éléments)

- En pseudo code :

variable **tableau** identificateur[**dimension**] : **type**

- Exemple :

variable **tableau** notes[**30**] : **réel**

- On peut définir des tableaux de tous types : tableaux d'entiers, de réels, de caractères, de booléens, de chaînes de caractères, ...

Tableaux : remarques

- L'accès à un élément du tableau se fait au moyen de l'indice. Par exemple, **notes[i]** donne la valeur de l'élément i du tableau notes
- Selon les langages, le premier indice du tableau est soit 0, soit 1. Le plus souvent c'est 0 (c'est ce qu'on va adopter en pseudo-code). Dans ce cas, **notes[i]** désigne l'élément i+1 du tableau notes
- Il est possible de déclarer un tableau sans préciser au départ sa dimension. Cette précision est faite ultérieurement
 - Par exemple, quand on déclare un tableau comme paramètre d'une procédure, on peut ne préciser sa dimension qu'au moment de l'appel
 - En tous cas, un tableau est inutilisable tant qu'on n'a pas précisé le nombre de ses éléments
- Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles

Tableaux : exemples (1)

- Pour le calcul du nombre d'étudiants ayant une note supérieure à 10 avec les tableaux, on peut écrire :

Variables i ,nbre : entier

tableau notes[30] : réel

Début

 nbre ← 0

Pour i allant de 0 à 29

Si (notes[i] >10) alors

 nbre ←nbre+1

FinSi

FinPour

 écrire ("le nombre de notes supérieures à 10 est : ", nbre)

Fin

Tableaux : saisie et affichage

- Procédures qui permettent de saisir et d'afficher les éléments d'un tableau :

Procédure SaisieTab(n : entier par valeur, **tableau** T : réel par référence)

variable i: entier

Pour i allant de 0 à n-1

 écrire ("Saisie de l'élément ", i + 1)

 lire (T[i])

FinPour

Fin Procédure

Procédure AfficheTab(n : entier par valeur, **tableau** T : réel par valeur)

variable i: entier

Pour i allant de 0 à n-1

 écrire ("T[,i, "] =", T[i])

FinPour

Fin Procédure

Tableaux : exemples d'appel

- Algorithme principale où on fait l'appel des procédures SaisieTab et AfficheTab :

Algorithme Tableaux

variable p : entier

tableau A[10] : réel

Début

 p ← 10

 SaisieTab(p, A)

 AfficheTab(10,A)

Fin

Tableaux : fonction longueur

La plus part des langages offrent une fonction **longueur** qui donne la dimension du tableau. Les procédures Saisie et Affiche peuvent être réécrites comme suit :

Procédure SaisieTab(**tableau** T : réel par référence)

variable i: entier

Pour i allant de 0 à **longueur**(T)-1

écrire ("Saisie de l'élément ", i + 1)

lire (T[i])

FinPour

Fin Procédure

Procédure AfficheTab(**tableau** T : réel par valeur)

variable i: entier

Pour i allant de 0 à **longueur**(T)-1

écrire ("T[,i, "] =", T[i])

FinPour

Fin Procédure

Tableaux : syntaxe Maple

- En Maple, un tableau se définit en utilisant le type **array** comme suit :

identificateur:= **array (a..b)**

- Identificateur est le nom du tableau
 - a et b sont les bornes de l'indice du tableau
- Il est possible d'entrer directement toutes les valeurs d'un tableau.

Exemple: **> A:=array(1..4,[5,8,1,7]);**

- Il est également possible de les entrer un par un comme suit :

Exemple : **> T:=array(1..3);**
 > T[1]:=1: T[2]:=3: T[3]:=5:

- Pour afficher tous les éléments d'un tableau, il suffit d'utiliser la commande print
 > print(T); ➡ **[1, 3, 5]**

Tableaux en malpe : exemple

- Une procédure qui calcule la moyenne des éléments d'un tableau :

```
> moyenne:=proc(n,T)
```

```
> local i,s;
```

```
> s:=0;
```

```
> for i from 1 to n do
```

```
> s:=s+T[i];
```

```
> od;
```

```
> s/n;
```

```
> end;
```

```
> A:=array(1..4,[5,8,1,7]);
```

```
> moyenne(4,A);
```

résultat : 21/4

Tableaux à deux dimensions

- Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par **deux indices**. Ceci est utile par exemple pour représenter des matrices
- En pseudo code, un tableau à deux dimensions se **déclare** ainsi :
variable tableau identificateur[dimension1] [dimension2] : type
 - Exemple : une matrice A de 3 lignes et 4 colonnes dont les éléments sont réels
variable tableau A[3][4] : réel
- **A[i][j]** permet d'accéder à l'élément de la matrice qui se trouve à l'intersection de la ligne i et de la colonne j

Exemples : lecture d'une matrice

- Procédure qui permet de saisir les éléments d'une matrice :

Procédure SaisieMatrice(n : entier par valeur, m : entier par valeur ,
tableau A : réel par référence)

Début

variables i, j : entier

Pour i allant de 0 à $n-1$

écrire ("saisie de la ligne ", $i + 1$)

Pour j allant de 0 à $m-1$

écrire ("Entrez l'élément de la ligne ", $i + 1$, " et de la colonne ", $j+1$)

lire (A[i][j])

FinPour

FinPour

Fin Procédure

Exemples : affichage d'une matrice

- Procédure qui permet d'afficher les éléments d'une matrice :

Procédure AfficheMatrice(n : entier par valeur, m : entier par valeur, **tableau** A : réel par valeur)

Début

variables i, j : entier

Pour i allant de 0 à $n-1$

Pour j allant de 0 à $m-1$

écrire ("A[",i, "]" ["",j,""]="", A[i][j])

FinPour

FinPour

Fin Procédure

Exemples : somme de deux matrices

- Procédure qui calcule la somme de deux matrices :

Procédure SommeMatrices(n, m : entier par valeur,
 tableau A, B : réel par valeur , **tableau** C : réel par référence)

Début

variables i, j : entier

Pour i allant de 0 à $n-1$

Pour j allant de 0 à $m-1$

$C[i][j] \leftarrow A[i][j] + B[i][j]$

FinPour

FinPour

Fin Procédure

Appel des procédures définies sur les matrices

Exemple d'algorithme principale où on fait l'appel des procédures définies précédemment pour la saisie, l'affichage et la somme des matrices :

Algorithme Matrices

variables **tableau** M1[3][4], M2 [3][4], M3 [3][4] : réel

Début

SaisieMatrice(3, 4, M1)

SaisieMatrice(3, 4, M2)

AfficheMatrice(3,4, M1)

AfficheMatrice(3,4, M2)

SommeMatrice(3, 4, M1,M2,M3)

AfficheMatrice(3,4, M3)

Fin

Matrices : syntaxe Maple

- Pour définir une matrice en Maple, on peut utiliser le type **array** ou le type **matrix** comme suit :

identificateur:= **array** (a1..b1, a2..b2)

identificateur:= **matrix**(n, m)

- a1 et b1 sont les bornes du premier indice du tableau
 - a2 et b2 sont les bornes du deuxième indice du tableau
 - n est le nombre de lignes et m le nombre de colonnes
- Il est possible d'entrer directement toutes les valeurs d'une matrice
Exemple: **> A:=matrix(2, 3, [[7,0,1], [2,4,3]]);**
- Le type **matrix** est disponible dans le **package linalg**. Il faut donc charger ce package avec la commande **with(linalg)** avant d'utiliser ce type

Tableaux : 2 problèmes classiques

- **Recherche d'un élément dans un tableau**
 - Recherche séquentielle
 - Recherche dichotomique
- **Tri d'un tableau**
 - Tri par sélection
 - Tri rapide

Recherche séquentielle

- Recherche de la valeur x dans un tableau T de N éléments :

Variables i: entier, Trouvé : booléen

...

i ← 0 , Trouvé ← Faux

TantQue (i < N) ET (Trouvé=Faux)

Si (T[i]=x) **alors**

 Trouvé ← Vrai

Sinon

 i ← i+1

FinSi

FinTantQue

Si Trouvé **alors** // c'est équivalent à écrire **Si** Trouvé=Vrai **alors**

écrire ("x appartient au tableau")

Sinon

écrire ("x n'appartient pas au tableau")

FinSi

Recherche séquentielle (version 2)

- Une fonction Recherche qui retourne un booléen pour indiquer si une valeur x appartient à un tableau T de dimension N .
 x , N et T sont des paramètres de la fonction

Fonction Recherche(x : réel, N : entier, tableau T : réel) : booléen

Variable i : entier

Pour i allant de 0 à $N-1$

Si ($T[i]=x$) **alors**

 retourne (Vrai)

FinSi

FinPour

 retourne (Faux)

FinFonction

Notion de complexité d'un algorithme

- Pour évaluer l'**efficacité** d'un algorithme, on calcule sa **complexité**
- Mesurer la **complexité** revient à quantifier le **temps** d'exécution et l'espace **mémoire** nécessaire
- Le temps d'exécution est proportionnel au **nombre des opérations** effectuées. Pour mesurer la complexité en temps, on met en évidence certaines opérations fondamentales, puis on les compte
- Le nombre d'opérations dépend généralement du **nombre de données** à traiter. Ainsi, la complexité est une fonction de la taille des données. On s'intéresse souvent à son **ordre de grandeur** asymptotique
- En général, on s'intéresse à la **complexité** dans le **pire des cas** et à la **complexité moyenne**

Recherche séquentielle : complexité

- Pour évaluer l'efficacité de l'algorithme de recherche séquentielle, on va calculer sa complexité dans le pire des cas. Pour cela on va compter le nombre de tests effectués
- Le pire des cas pour cet algorithme correspond au cas où x n'est pas dans le tableau T
- Si x n'est pas dans le tableau, on effectue $3N$ tests : on répète N fois les tests ($i < N$), (Trouvé=Faux) et ($T[i]=x$)
- La **complexité** dans le pire des cas est **d'ordre N** , (on note **$O(N)$**)
- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1ms	1s	16mn40s

Recherche dichotomique

- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique
- **Principe** : diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur x à chaque étape de la recherche. Pour cela on compare x avec $T[\text{milieu}]$:
 - Si $x < T[\text{milieu}]$, il suffit de chercher x dans la 1ère moitié du tableau entre ($T[0]$ et $T[\text{milieu}-1]$)
 - Si $x > T[\text{milieu}]$, il suffit de chercher x dans la 2ème moitié du tableau entre ($T[\text{milieu}+1]$ et $T[N-1]$)

Recherche dichotomique : algorithme

inf ← 0 , sup ← N-1, Trouvé ← Faux

TantQue (inf ≤ sup) ET (Trouvé = Faux)

milieu ← (inf + sup) div 2

Si (x = T[milieu]) **alors**

Trouvé ← Vrai

Sinon Si (x > T[milieu]) **alors**

inf ← milieu + 1

Sinon sup ← milieu - 1

FinSi

FinSi

FinTantQue

Si Trouvé **alors** écrire ("x appartient au tableau")

Sinon écrire ("x n'appartient pas au tableau")

FinSi

Exemple d'exécution

- Considérons le tableau T :

4	6	10	15	17	18	24	27	30
---	---	----	----	----	----	----	----	----

- Si la valeur cherché est 20 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	5	5	6
sup	8	8	5	5
milieu	4	6	5	

- Si la valeur cherché est 10 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	0	2
sup	8	3	3
milieu	4	1	2

Recherche dichotomique : complexité

- La complexité dans le pire des cas est d'ordre $\log_2 N$
- L'écart de performances entre la recherche séquentielle et la recherche dichotomique est considérable pour les grandes valeurs de N
 - Exemple: au lieu de $N=1\text{million} \approx 2^{20}$ opérations à effectuer avec une recherche séquentielle il suffit de 20 opérations avec une recherche dichotomique

Tri d'un tableau

- Le tri consiste à ordonner les éléments du tableau dans l'ordre croissant ou décroissant
- Il existe plusieurs algorithmes connus pour trier les éléments d'un tableau :
 - Le tri par sélection
 - Le tri par insertion
 - Le tri rapide
 - ...
- Nous verrons dans la suite l'algorithme de tri par sélection et l'algorithme de tri rapide. Le tri sera effectué dans l'ordre croissant

Tri par sélection

- **Principe** : à l'étape i , on sélectionne le plus petit élément parmi les $(n - i + 1)$ éléments du tableau les plus à droite. On l'échange ensuite avec l'élément i du tableau

- **Exemple :**

9	4	1	7	3
---	---	---	---	---

- **Étape 1:** on cherche le plus petit parmi les 5 éléments du tableau. On l'identifie en troisième position, et on l'échange alors avec l'élément 1 :

1	4	9	7	3
---	---	---	---	---

- **Étape 2:** on cherche le plus petit élément, mais cette fois à partir du deuxième élément. On le trouve en dernière position, on l'échange avec le deuxième:

1	3	9	7	4
---	---	---	---	---

- **Étape 3:**

1	3	4	7	9
---	---	---	---	---

Tri par sélection : algorithme

- Supposons que le tableau est noté T et sa taille N

Pour i allant de 0 à $N-2$

$\text{indice_ppe} \leftarrow i$

Pour j allant de $i + 1$ à $N-1$

Si $T[j] < T[\text{indice_ppe}]$ **alors**

$\text{indice_ppe} \leftarrow j$

Finsi

FinPour

$\text{temp} \leftarrow T[\text{indice_ppe}]$

$T[\text{indice_ppe}] \leftarrow T[i]$

$T[i] \leftarrow \text{temp}$

FinPour

Tri par sélection : complexité

- Quel que soit l'ordre du tableau initial, le nombre de tests et d'échanges reste le même
- On effectue $N-1$ tests pour trouver le premier élément du tableau trié, $N-2$ tests pour le deuxième, et ainsi de suite. Soit : $(N-1)+(N-2)+\dots+1 = N(N-1)/2$
On effectue en plus $(N-1)$ échanges.
- La **complexité** du tri par sélection est **d'ordre N^2** à la fois dans le meilleur des cas, en moyenne et dans le pire des cas
- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1s	11,5 jours	32000 ans

Tri rapide

- Le tri rapide est un tri récursif basé sur l'approche "diviser pour régner" (consiste à décomposer un problème d'une taille donnée à des sous problèmes similaires mais de taille inférieure faciles à résoudre)
- **Description du tri rapide :**
 - 1) on considère un élément du tableau qu'on appelle pivot
 - 2) on partitionne le tableau en 2 sous tableaux : les éléments inférieurs ou égaux à pivot et les éléments supérieurs à pivot. on peut placer ainsi la valeur du pivot à sa place définitive entre les deux sous tableaux
 - 3) on répète récursivement ce partitionnement sur chacun des sous tableaux créés jusqu'à ce qu'ils soient réduits à un à un seul élément

Procédure Tri rapide

Procédure TriRapide(tableau **T** : réel par adresse, **p,r**: entier par valeur)

variable **q**: entier

Si $p < r$ **alors**

 Partition(**T**,**p**,**r**,**q**)

 TriRapide(**T**,**p**,**q**-1)

 TriRapide(**T**,**q**+1,**r**)

FinSi

Fin Procédure

A chaque étape de récursivité on partitionne un tableau $T[p..r]$ en deux sous tableaux $T[p..q-1]$ et $T[q+1..r]$ tel que chaque élément de $T[p..q-1]$ soit inférieur ou égal à chaque élément de $T[q+1..r]$. L'indice q est calculé pendant la procédure de partitionnement

Procédure de partition

Procédure Partition(tableau **T** : réel par adresse, **p,r**: entier par valeur,
q: entier par adresse)

Variables **i, j**: entier

pivot: réel

pivot \leftarrow T[p], **i** \leftarrow p+1, **j** \leftarrow r

TantQue (**i** \leq **j**)

TantQue (**i** \leq r et T[**i**] \leq pivot) **i** \leftarrow **i**+1 **FinTantQue**

TantQue (**j** \geq p et T[**j**] $>$ pivot) **j** \leftarrow **j**-1 **FinTantQue**

Si **i** $<$ **j** **alors**

Echanger(T[**i**], T[**j**]), **i** \leftarrow **i**+1, **j** \leftarrow **j**-1

FinSi

FinTantQue

Echanger(T[**j**], T[p])

q \leftarrow **j**

Fin Procédure

Tri rapide : complexité et remarques

- La complexité du tri rapide dans le pire des cas est en $O(N^2)$
- La complexité du tri rapide en moyenne est en $O(N \log N)$
- Le choix du pivot influence largement les performances du tri rapide
- Le pire des cas correspond au cas où le pivot est à chaque choix le plus petit élément du tableau (tableau déjà trié)
- différentes versions du tri rapide sont proposés dans la littérature pour rendre le pire des cas le plus improbable possible, ce qui rend cette méthode la plus rapide en moyenne parmi toutes celles utilisées